说说 JAVA 代理模式

ImportNew *2017-08-19*

来源：姜肇海 投稿，
www.importnew.com/26116.html
如有好文章投稿，请点击 → 这里了解详情

事例

小张是一个普普通通的码农，每天勤勤恳恳地码代码。某天中午小张刚要去吃饭，一个电话打到了他的手机上。“是XX公司的小张吗？我是YY公司的王AA”。“哦，是王总啊，有什么事情吗？”。沟通过后，小张弄明白了,原来客户有个需求，刚好负责这方面开发的是小张，客户就直接找到了他。不过小张却没有答应客户的请求，而是让客户找产品经理小李沟通。

是小张着急去吃面而甩锅吗？并不是，只是为了使故事可以套到代理模式上。我们先看一下代理模式的定义： * 为其他对象提供一种代理，以控制对这个对象的访问。(Provide a surrogate or placeholder for another object to control access to it)

对照定义，码农小张可以映射为其他对象，产品经理小李为小张的代理。我们通过JAVA代码，表述上面事例。

静态代理

1.抽象角色

基于面向对象的思想，首先定义一个码农接口,它有一个实现用户需求的方法。

```
public interface ICoder {

    public void implDemands(String demandName);

}
```

2.真实角色

我们假设小张是JAVA程序员，定义一个JAVA码农类，他通过JAA语言实现需求。

```
public class JavaCoder implements ICoder{

    private String name;
```

```
    public JavaCoder(String name){
        this.name = name;
    }

    @Override
    public void implDemands(String demandName) {
        System.out.println(name + " implemented demand:" + demandName + " in JAVA!");
    }
}
```

3.代理角色

委屈一下产品经理，将其命名为码农代理类，同时让他实现ICoder接口。

```
public class CoderProxy implements ICoder{

    private ICoder coder;

    public CoderProxy(ICoder coder){
        this.coder = coder;
    }

    @Override
    public void implDemands(String demandName) {
        coder.implDemands(demandName);
    }
}
```

上面一个接口，两个类，就实现了代理模式。Are you kidding me？这么简单？是的，就是这么简单。我们通过一个场景类，模拟用户找产品经理增加需求。

```
public class Customer {

    public static void main(String args[]){
        //定义一个java码农
        ICoder coder = new JavaCoder("Zhang");
        //定义一个产品经理
        ICoder proxy = new CoderProxy(coder);
        //让产品经理实现一个需求
        proxy.implDemands();
    }
}
```

运行程序，结果如下：

```
Zhang implemented demand:Add user manageMent in JAVA!
```

产品经理充当了程序员的代理，客户把需求告诉产品经理，并不需要和程序员接触。看到这里，有些机智的程序员发现了问题。你看，产品经理就把客户的需求转达了一下，怪不得我看产品经理这么不爽。

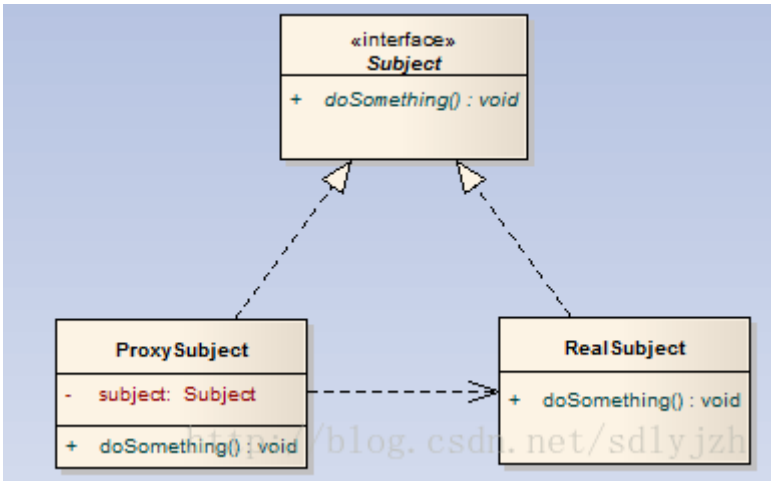
产品经理当然不只是转达用户需求，他还有很多事情可以做。比如，该项目决定不接受新增功能的需求了，对修CoderProxy类做一些修改：

```
public class CoderProxy implements ICoder{

    private ICoder coder;

    public CoderProxy(ICoder coder){
        this.coder = coder;
    }

    @Override
    public void implDemands(String demandName) {
        if(demandName.startsWith("Add")){
            System.out.println("No longer receive 'Add' demand");
            return;
        }
        coder.implDemands(demandName);
    }
}
```

这样，当客户再有增加功能的需求时，产品经理就直接回绝了，程序员无需再对这部分需求做过滤。

总结

我们对上面的事例做一个简单的抽象：



代理模式包含如下角色：

- Subject:抽象主题角色。可以是接口，也可以是抽象类。
- RealSubject:真实主题角色。业务逻辑的具体执行者。
- ProxySubject:代理主题角色。内部含有RealSubject的引用,负责对真实角色的调用，并在真实主题角色处理前后做预处理和善后工作。

代理模式优点：

- 职责清晰 真实角色只需关注业务逻辑的实现，非业务逻辑部分，后期通过代理类完成即可。
- 高扩展性 不管真实角色如何变化，由于接口是固定的，代理类无需做任何改动。

动态代理

前面讲的主要是静态代理。那么什么是动态代理呢？

假设有这么一个需求，在方法执行前和执行完成后，打印系统时间。这很简单嘛，非业务逻辑，只要在代理类调用真实角色的方法前、后输出时间就可以了。像上例，只有一个implDemands方法，这样实现没有问题。但如果真实角色有10个方法，那么我们要写10遍完全相同的代码。有点追求的码农，肯定会对这种方法感到非常不爽。有些机智的小伙伴可能想到了用AOP解决这个问题。非常正确。莫非AOP和动态代理有什么关系？没错！AOP用的恰恰是动态代理。

代理类在程序运行时创建的代理方式被称为动态代理。也就是说，代理类并不需要在Java代码中定义，而是在运行时动态生成的。相比于静态代理， 动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类的函数。对于上例打印时间的需求，通过使用动态代理，我们可以做一个“统一指示”，对所有代理类的方法进行统一处理，而不用逐一修改每个方法。下面我们来具体介绍下如何使用动态代理方式实现我们的需求。

与静态代理相比，抽象角色、真实角色都没有变化。变化的只有代理类。因此，抽象角色、真实角色，参考ICoder和JavaCodr。

在使用动态代理时，我们需要定义一个位于代理类与委托类之间的中介类，也叫动态代理类，这个类被要求实现InvocationHandler接口：

```
public class CoderDynamicProxy implements InvocationHandler{
    //被代理的实例
    private ICoder coder;

    public CoderDynamicProxy(ICoder _coder){
        this.coder = _coder;
    }
```

```
    //调用被代理的方法
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println(System.currentTimeMillis());
        Object result = method.invoke(coder, args);
        System.out.println(System.currentTimeMillis());
        return result;
    }
}
```

当我们调用代理类对象的方法时，这个“调用”会转送到中介类的invoke方法中，参数method标识了我们具体调用的是代理类的哪个方法，args为这个方法的参数。

我们通过一个场景类，模拟用户找产品经理更改需求。

```
public class DynamicClient {

    public static void main(String args[]){
        //要代理的真实对象
        ICoder coder = new JavaCoder("Zhang");
        //创建中介类实例
        InvocationHandler  handler = new CoderDynamicProxy(coder);
        //获取类加载器
        ClassLoader cl = coder.getClass().getClassLoader();
        //动态产生一个代理类
        ICoder proxy = (ICoder) Proxy.newProxyInstance(cl, coder.getClass().getInterfaces(), handler);
        //通过代理类，执行doSomething方法；
        proxy.implDemands("Modify user management");
    }
}
```

执行结果如下：

```
1501728574978
Zhang implemented demand:Modify user management in JAVA!
1501728574979
```

通过上述代码，就实现了，在执行委托类的所有方法前、后打印时间。还是那个熟悉的小张，但我们并没有创建代理类，也没有时间ICoder接口。这就是动态代理。

总结

总结一下，一个典型的动态代理可分为以下四个步骤：

6. 创建抽象角色
7. 创建真实角色
8. 通过实现InvocationHandler接口创建中介类
9. 通过场景类，动态生成代理类

如果只是想用动态代理，看到这里就够了。但如果想知道为什么通过proxy对象，就能够执行中介类的invoke方法，以及生成的proxy对象是什么样的，可以继续往下看。

源码分析(JDK7)

看到这里的小伙伴，都是有追求的程序员。上面的场景类中，通过

```
//动态产生一个代理类
ICoder proxy = (ICoder) Proxy.newProxyInstance(cl, coder.getClass().getInterfaces(), handler);
```

动态产生了一个代理类。那么这个代理类是如何产生的呢？我们通过代码一窥究竟。

Proxy类的newProxyInstance方法，主要业务逻辑如下：

```
//生成代理类class，并加载到jvm中
Class<?> cl = getProxyClass0(loader, interfaces);
//获取代理类参数为InvocationHandler的构造函数
final Constructor<?> cons = cl.getConstructor(constructorParams);
//生成代理类，并返回
return newInstance(cons, ih);
```

上面代码做了三件事：

- 根据传入的参数interfaces动态生成一个类，它实现interfaces中的接口，该例中即ICoder接口的implDemands方法。假设动态生成的类为$Proxy0。
- 通过传入的classloder,将刚生成的$Proxy0类加载到jvm中。
- 利用中介类，调用$Proxy0的$Proxy0(InvocationHandler)构造函数，创建$Proxy0类的实例，其InvocationHandler属性，为我们创建的中介类。

上面的核心，就在于getProxyClass0方法：

```
private static Class<?> getProxyClass0(ClassLoader loader,
                                       Class<?>... interfaces) {
    if (interfaces.length > 65535) {
        throw new IllegalArgumentException("interface limit exceeded");
    }
```

```
        // If the proxy class defined by the given loader implementing
        // the given interfaces exists, this will simply return the cached copy;
        // otherwise, it will create the proxy class via the ProxyClassFactory
        return proxyClassCache.get(loader, interfaces);
    }
```

在Proxy类中有个属性proxyClassCache，这是一个WeakCache类型的静态变量。它指示了类加载器和代理类之间的映射。所以proxyClassCache的get方法用于根据类加载器来获取Proxy类，如果已经存在则直接从cache中返回，如果没有则创建一个映射并更新cache表。

我们跟一下代理类的创建流程：
调用Factory类的get方法，而它又调用了ProxyClassFactory类的apply方法，最终找到下面一行代码：

```
//Generate the specified proxy class.
byte[] proxyClassFile = ProxyGenerator.generateProxyClass(proxyName, interfaces);
```

就是它，生成了代理类。

查看动态生成的代理类

通过上面的分析，我们已经知道Proxy类动态创建代理类的流程。那创建出来的代理类到底是什么样子的呢？我们可以通过下面的代码，手动生成：

```
public class CodeUtil {

    public static void main(String[] args) throws IOException {
        byte[] classFile = ProxyGenerator.generateProxyClass("TestProxyGen",
JavaCoder.class.getInterfaces());
        File file = new File("D:/aaa/TestProxyGen.class");
        FileOutputStream fos = new FileOutputStream(file);
        fos.write(classFile);
        fos.flush();
        fos.close();
      }
}
```

通过反编译工具查看生成的class文件:

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.lang.reflect.UndeclaredThrowableException;
import model.proxy.ICoder;
```

```
public final class TestProxyGen extends Proxy
  implements ICoder
{
  private static Method m1;
  private static Method m0;
  private static Method m3;
  private static Method m2;

  public TestProxyGen(InvocationHandler paramInvocationHandler)
    throws
  {
    super(paramInvocationHandler);
  }

  public final boolean equals(Object paramObject)
    throws
  {
    try
    {
      return ((Boolean)this.h.invoke(this, m1, new Object[] { paramObject })).booleanValue();
    }
    catch (RuntimeException localRuntimeException)
    {
      throw localRuntimeException;
    }
    catch (Throwable localThrowable)
    {
    }
    throw new UndeclaredThrowableException(localThrowable);
  }

  public final int hashCode()
    throws
  {
    try
    {
      return ((Integer)this.h.invoke(this, m0, null)).intValue();
    }
    catch (RuntimeException localRuntimeException)
    {
      throw localRuntimeException;
    }
    catch (Throwable localThrowable)
    {
    }
    throw new UndeclaredThrowableException(localThrowable);
```

```
  }

  public final void implDemands(String paramString)
    throws
  {
    try
    {
      this.h.invoke(this, m3, new Object[] { paramString });
      return;
    }
    catch (RuntimeException localRuntimeException)
    {
      throw localRuntimeException;
    }
    catch (Throwable localThrowable)
    {
    }
    throw new UndeclaredThrowableException(localThrowable);
  }

  public final String toString()
    throws
  {
    try
    {
      return (String)this.h.invoke(this, m2, null);
    }
    catch (RuntimeException localRuntimeException)
    {
      throw localRuntimeException;
    }
    catch (Throwable localThrowable)
    {
    }
    throw new UndeclaredThrowableException(localThrowable);
  }

  static
  {
    try
    {
      m1 = Class.forName("java.lang.Object").getMethod("equals", new Class[] {
Class.forName("java.lang.Object") });
      m0 = Class.forName("java.lang.Object").getMethod("hashCode", new Class[0]);
      m3 = Class.forName("model.proxy.ICoder").getMethod("implDemands", new Class[] {
Class.forName("java.lang.String") });
      m2 = Class.forName("java.lang.Object").getMethod("toString", new Class[0]);
```

```
        return;
      }
      catch (NoSuchMethodException localNoSuchMethodException)
      {
        throw new NoSuchMethodError(localNoSuchMethodException.getMessage());
      }
      catch (ClassNotFoundException localClassNotFoundException)
      {
      }
      throw new NoClassDefFoundError(localClassNotFoundException.getMessage());
    }
  }
```

这样，我们就理解，为什么调用代理类的implDemands方法，回去执行中介类的invoke方法了。

Read more